OpenJML and SMT solvers

Leonidas Tsiopoulos

ITI8610 Software Assurance Course, Module II, Lecture 4 - 15/11/2018

Precursors to OpenJML

- JML was first used in an early *extended static checker* (ESC/Java) and was implemented in a set of tools called JML2.

- The second generation of ESC/Java, ESC/Java2, was made current with Java 1.4 and with the definition of JML.

- JML2 tools were based on hand-crafted compilers and the maintenance and update effort was overwhelming as Java evolved.

- A new approach was needed, one that built on an existing compiler to leverage further developments in that compiler but *allowed easy integration* with a Java IDE environment, and was *readily maintainable and extensible*.

OpenJML - Introduction

- OpenJML is an implementation of JML tools built by extending the OpenJDK Java tool set.
- OpenJDK has a readily extensible architecture, although it is quite amenable to extension since it has a complex compilation process with many components.
- The result is a suite of JML tools for **Java 8** that provides *static analysis*, *specification documentation*, and *runtime checking*, an API that is used for other tools, uses Eclipse as an IDE, and can be extended for further research.
- The main drawback is that in an Eclipse-integrated system, the Eclipse compiler is used (as is) for Java compilation and the OpenJML/OpenJDK compiler is used as a back-end tool for handling JML and verification tasks.

OpenJML command-line tool

- Ability to parse and type-check current JML
- Ability to perform static verification checks using back-end SMT solvers
- Ability to explore counterexamples (models) provided by the solver
- Partial implementation of JML-aware documentation generation
- Proof of concept implementation of runtime assertion checking
- JMLUnitNG has used OpenJML to create a test generation tool, using OpenJML's API to access the parsed specifications

Eclipse Java development environment OpenJML plug-in

- Ability to parse and type-check JML showing any errors or warnings as Eclipse problems, but with a custom icon and problem type
- Ability to check JML specifications against the Java code
  - Verification conditions are produced from the internal **ASTs** (Abstract Syntax Trees) and submitted to a back-end **Satisfiability Modulo Theories (SMT)** solver, and any proof failures are shown as Eclipse problems.
- Ability to use files with runtime checks along with Eclipse-compiled files
- Ability to explore specifications and counterexamples *within* the GUI
- Functionality integrated as Eclipse menus, commands, and editor windows

Exploring Counterexamples from Static Checking

- The Eclipse GUI enables exploring counterexamples produced by failed static checking much more effectively than previous JML tools.

- The Eclipse GUI for OpenJML interprets the counterexample information and relates it directly to the program as seen in the Eclipse editor windows.

- Previously, other tools created verification conditions, shipped them to a back-end solver, which produced counterexample information that was essentially a dump of the prover state and was notoriously difficult to debug.

OpenJML back-end SMT solvers – What is SMT?

- SMT solvers are useful for verification, proving the correctness of programs, software testing based on symbolic execution, and for program synthesis.

- Computer-aided verification of computer programs often uses SMT solvers.

- In computer science and mathematical logic, the **Satisfiability Modulo Theories** (**SMT**) problem is a decision problem for logical formulas *with respect to combinations of background theories* expressed in classical *first-order logic with equality.*
    - Examples of theories: Real numbers, integers, theories of data structures like lists, arrays, bit-vectors, ...

SMT Instances

- An SMT instance is a *formula in first-order logic*, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable.
- An SMT instance is a *generalization* of a *Boolean SAT instance* in which various sets of variables are replaced by predicates from corresponding underlying theories.
  - **Boolean SAT problem** is the problem of determining if there exists an interpretation that satisfies a given Boolean formula, i.e., it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.
    - E.g., "*a* AND NOT *b*" is satisfiable and "*a* AND NOT *a*" is unsatisfiable.
- SMT formulas provide a much richer modeling language than is possible with Boolean SAT formulas.

Verification and Testing with SMT Solvers

- For verification of programs a common technique is to translate *pre-conditions*, *post-conditions*, *loop conditions*, and *assertions* into SMT formulas in order to determine if all properties can hold.

- Another important application of SMT solvers is *symbolic execution* for *analysis* and *testing* of programs.

OpenJML back-end SMT solvers

- OpenJML translates JML specifications into SMT-LIB format and passes the proof problems implied by the Java+JML program to back-end SMT solvers.

- OpenJML can use any SMT-LIBv2-compliant solver.
  - Z3, CVC4, Yices, ...
  - *Simplify* was the theorem prover of the *Extended Static Checkers* ESC/Java and still supported by OpenJML.

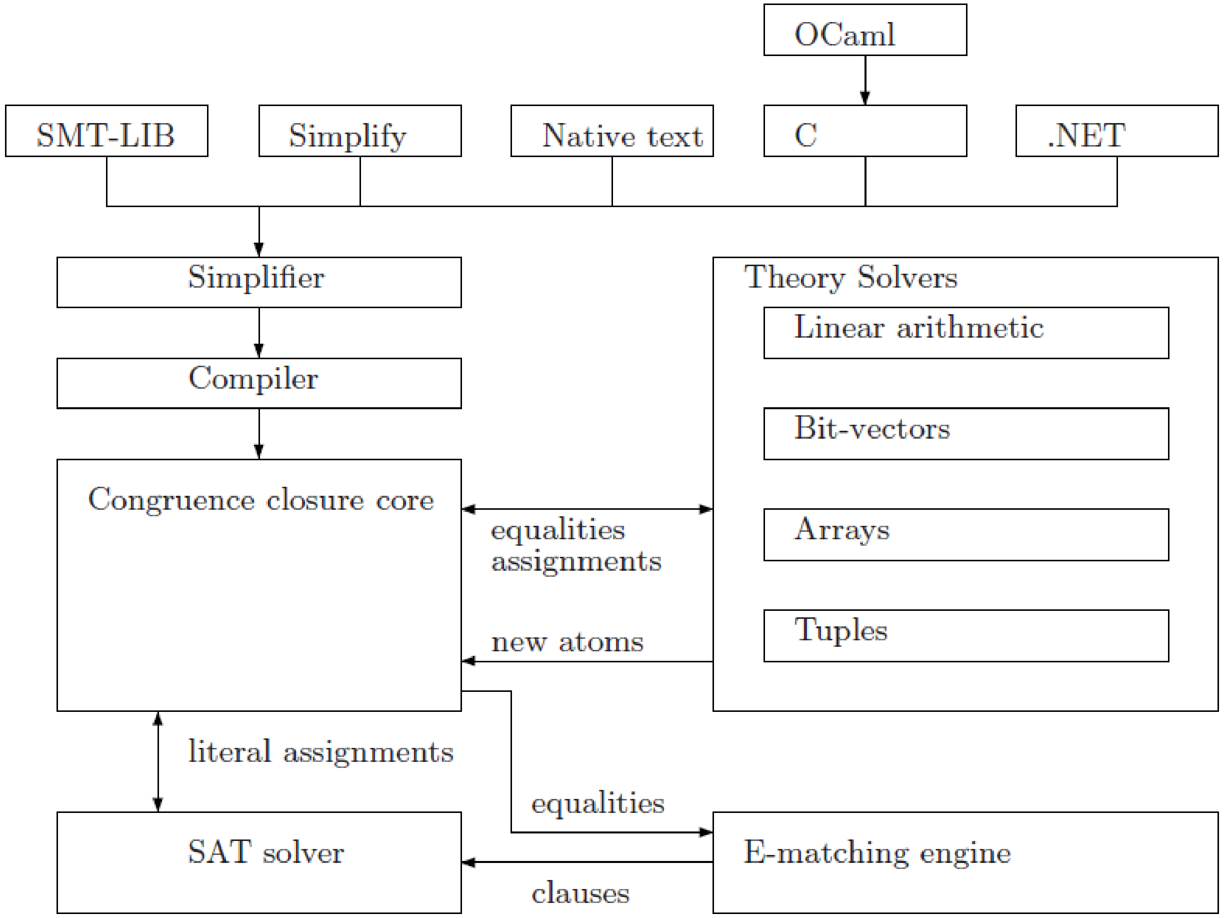
- Success in checking the consistency of the specifications and the code will depend on:
  - (a) the capability of the back-end SMT solver,
  - (b) the particular encoding of code + specifications into SMT by OpenJML, and
  - (c) the complexity and style in which the code and specifications are written.

OpenJML Z3 back-end SMT solver

- Supported theories: *empty theory*, *linear arithmetic*, *nonlinear arithmetic*, *bit-vectors*, *arrays*, *datatypes*, *quantifiers*, *strings*
- Advanced algorithms for quantifier instantiation and theory combination.
- Z3 integrates a DPLL-based SAT solver, a core theory solver for equalities and uninterpreted functions, *satellite solvers* and an engine for quantifiers.

To get started:

https://rise4fun.com/z3/tutorial/guide



OpenJML CVC4 back-end SMT solver

- CVC4 works with a version of first-order logic with ***polymorphic*** types.
  - http://cvc4.cs.stanford.edu/web/

- Several built-in base theories: rational and integer linear arithmetic, arrays, tuples, records, inductive data types, bit-vectors, strings, and equality over uninterpreted function symbols (“empty theory”).

- Support for quantifiers through heuristic instantiation.

- CVC4 is fundamentally similar to other modern SMT solvers like Z3: it is a DPLL solver, with a SAT solver at its core and a delegation path to different decision procedure implementations, each in charge of solving formulas in some background theory.

OpenJML and Simplify theorem prover

- *Simplify* is a theorem prover for program checking developed at HP Labs.
- Simplify is the proof engine of the *Extended Static Checkers* ESC/Java.
- The goal of ESC is to prove, at compile-time, the absence of certain run-time errors, such as out-of-bounds array accesses and unhandled exceptions.
- The ESC approach first processes source code with a verification condition generator, which produces first-order formulas asserting the *absence* of the targeted errors, and then submits those verification conditions to the theorem prover.

OpenJML and Simplify theorem prover (cont.)

- Input to *Simplify* is an arbitrary first-order formula, including quantifiers.

- *Simplify* handles propositional connectives by *backtracking search* and includes complete decision procedures for the supported theories (untyped first-order logic with function symbols and equality, arithmetic, maps, partial orders, ...).

- To test whether a formula is satisfiable, *Simplify* performs a backtracking search, guided by the propositional structure of the formula, attempting to find a satisfying assignment of truth values to atomic formulas that makes the formula *true*.

OpenJML Yices 2 back-end SMT solver

- Yices 2[1] is an SMT solver that decides the satisfiability of formulas containing uninterpreted function symbols with equality, real and integer arithmetic, bit-vectors, scalar types, and tuples.

- Both linear and nonlinear arithmetic is supported.

- Yices 2 includes a congruence-closure algorithm inspired by Simplify's E-graph and used an approach for theory combination based on the Nelson-Oppen method (also used in Simplify and other SMT solvers like Z3) complemented with lazy generation of interface equalities.

[1] http://yices.csl.sri.com/

OpenJML and Testing

- **JMLUnitNG**[1] is an automated unit test generation tool for JML-annotated Java code, including code using Java 1.5+ features such as generics, enumerated types, and enhanced for loops.
- JML assertions are used as *test oracles*.
- Tests can be generated for OpenJML RAC.
- Testing a class (or set of classes) with JMLUnitNG involves:
    1. Generating the test classes
    2. Compile the classes under test with OpenJML
    3. Compile the generated (test) classes with a regular Java compiler
    4. Run the tests.

[1] http://insttech.secretninjaformalmethods.org/software/jmlunitng/

The following slides are based on material presented by Leonardo de Moura and Nikolaj Bjørner in various presentations on Z3, found on:

http://leodemoura.github.io/slides.html

Basics - Language of logic

- Functions , Variables, Predicates
  - *f, g,* *x, y, z,* P, Q, =
- Atomic formulas, Literals
  - $P(x,f(y)), \neg Q(y,z)$
- Quantifier free formulas
  - $P(f(a), b) \wedge c = g(d)$
- Formulas, sentences
  - $\forall x . \forall y . [P(x, f(x)) \vee g(y,x) = h(y)]$

Language: Signatures

- A *signature* Σ is a finite set of:
  - Function symbols:

    Σ_F = { *f*, *g*, … }
  - Predicate symbols:

    Σ_P = { *P*, *Q*,=, true, false, … }
  - And an *arity* function:

    $\Sigma \rightarrow N$
- Function symbols with arity 0 are *constants*
- A countable set *V* of *variables*
  - disjoint from Σ

Language: Quantifier free formulas

- The set QFF(Σ,V) of *quantifier free formulas* is the smallest set such that:

$\varphi \in$ QFF	$::= a \in Atoms$	*atoms*
	$\mid \neg \varphi$	*negations*
	$\mid \varphi \leftrightarrow \varphi'$	*bi-implications*
	$\mid \varphi \wedge \varphi'$	*conjunction*
	$\mid \varphi \vee \varphi'$	*disjunction*
	$\mid \varphi \rightarrow \varphi'$	*implication*

Language: Formulas

- The set of *first-order formulas* are obtained by adding the formation rules:

$$
\begin{array}{llll}
\varphi ::= \ldots & & & \\
& \mid & \forall\, x \,.\, \varphi & \textit{universal quant.} \\
& \mid & \exists\, x \,.\, \varphi & \textit{existential quant.}
\end{array}
$$

- *Free* (occurrences) of *variables* in a formula are those not bound by a quantifier.
- *A sentence* is a first-order formula with no free variables.

Theories

- A (first-order) *theory T (over signature Σ) is a set of (deductively closed) sentenes (over Σ and V)*

- Let *DC*(Γ) be the *deductive closure* of a set of sentences Γ.
  - For every theory T, *DC*(T) = T

- A theory T is *constistent* if *false* $\notin$ *T*

- We can view a (first-order) theory *T* as the class of all *models* of *T* (due to completeness of first-order logic).

Models (Semantics)

- A model M is defined as:
  - Domain S; set of elements.
  - Interpretation, $f^M : S^n \rightarrow S$ for each $f \in \Sigma_F$ with arity(f) = n
  - Interpretation $P^M \subseteq S^n$ for each $P \in \Sigma_P$ with arity(P) = n
  - Assignment $x^M \in S$ for every variable $x \in V$

- *A formula* φ is true in a model M if it evaluates to true under the given interpretations over the domain S.

- M is a *model for the theory T* if all sentences of T are true in M.

T-Satisfiability

- A formula $\varphi(x)$ is T-satisfiable in a theory T if there is a model of $DC(T \cup \exists\, x\ \varphi(x))$. That is, there is a model M for T in which $\varphi(x)$ evaluates to true.

- Notation:

$$M \vDash_T \varphi(x)$$

T-Validity

- A formula $\varphi(x)$ is T-*valid* in a theory T if $\forall\ x\ \varphi(x) \in T$. That is, $\varphi(x)$ evaluates to *true* in every model M of T.

- *T-validity:*

$$\vDash_{T} \varphi(x)$$

Checking Validity – the morale

- Theory solvers must minimally be able to:
  - check *unsatisfiability* of conjunctions of literals.

Clauses – CNF conversion

Generally SMT solvers work with formulas in *Conjunctive Normal Form* (CNF).

$\varphi : x = 5 \Leftrightarrow (y < 3 \vee z = x)$ *is not in CNF.*

Clauses – CNF conversion

$$\varphi : x = 5 \Leftrightarrow (y < 3 \vee z = x)$$

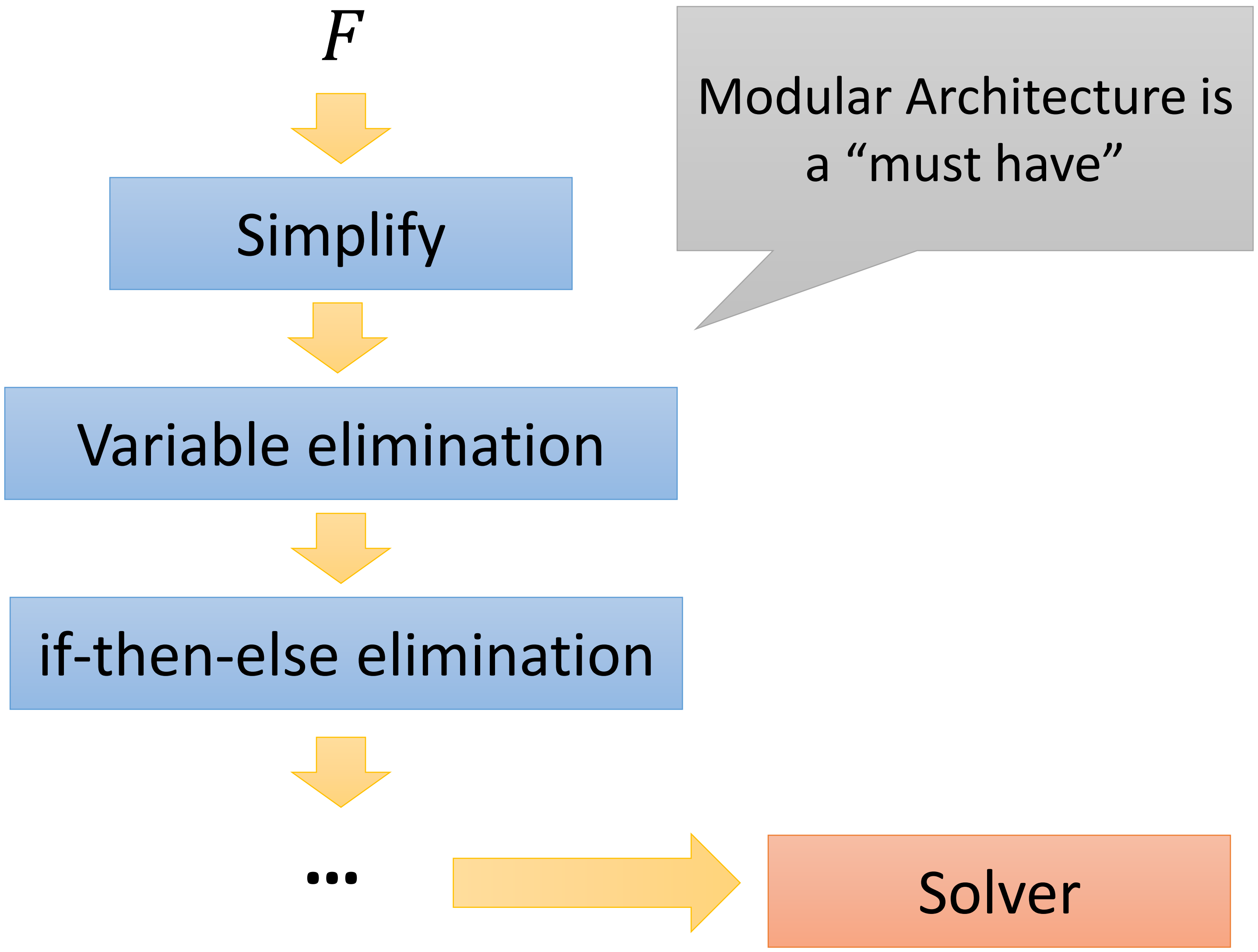
Equi-satisfiable CNF formula

$$\varphi' : (\neg p \vee x = 5) \wedge (p \vee \neg x = 5) \wedge$$
$$(\neg p \vee y < 3 \vee z = x) \wedge$$
$$(p \vee \neg y < 3) \wedge (p \vee \neg z = x)$$

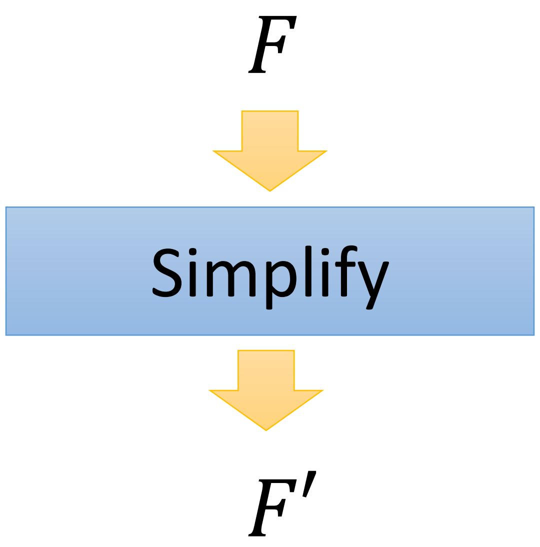
Clauses - CNF

- Main properties of basic CNF:
  - Result *F* is a set of *clauses.*
  - φ is *T*-satisfiable iff CNF(φ) is.
  - size(CNF(φ)) $\leq$ 4(size(φ))
  - $\varphi \Leftrightarrow \exists\, p_{aux}$ CNF(φ)

Preprocessing of formulas for SMT solver



Equivalence Preserving Simplifications

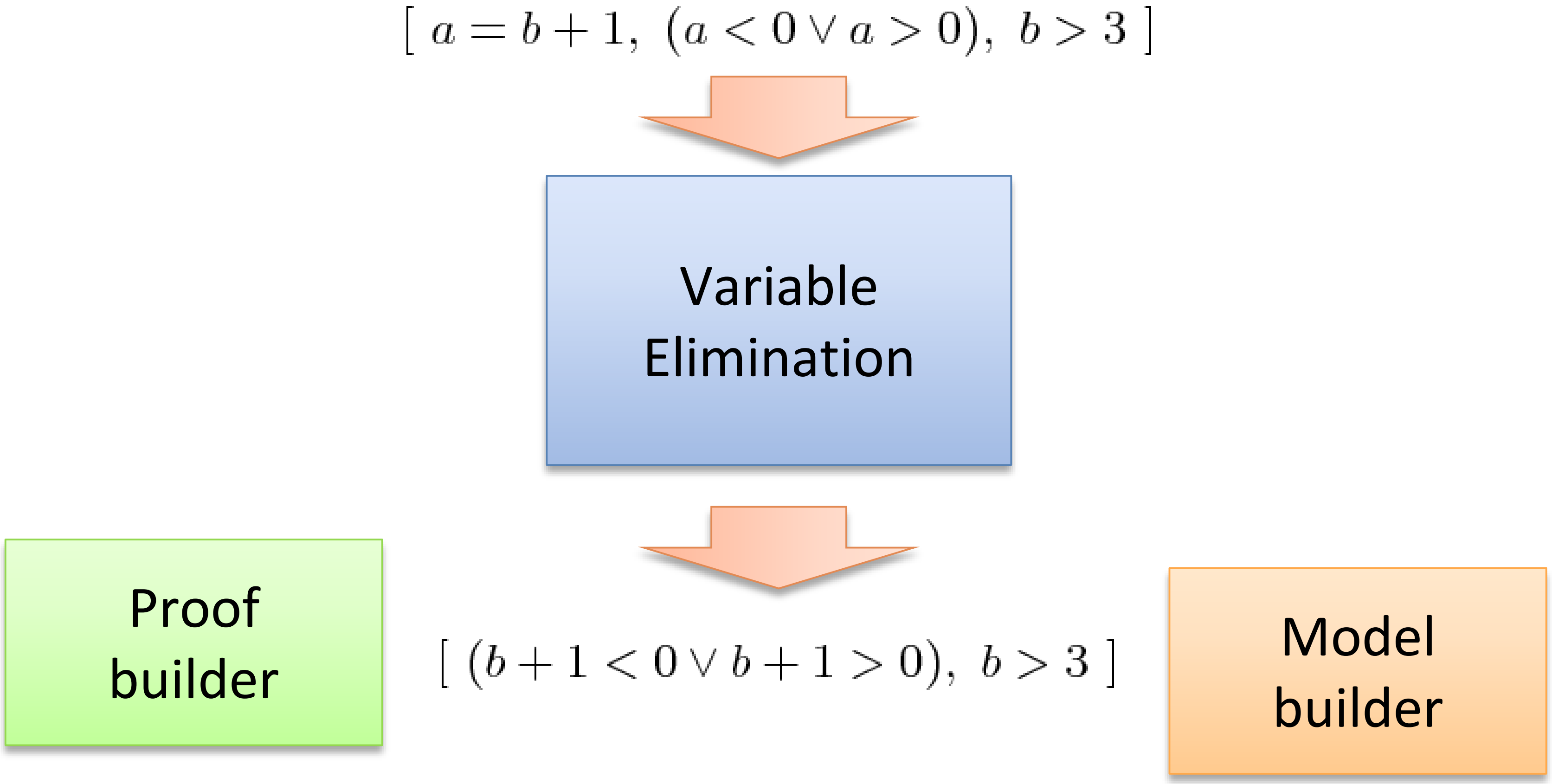


Examples:

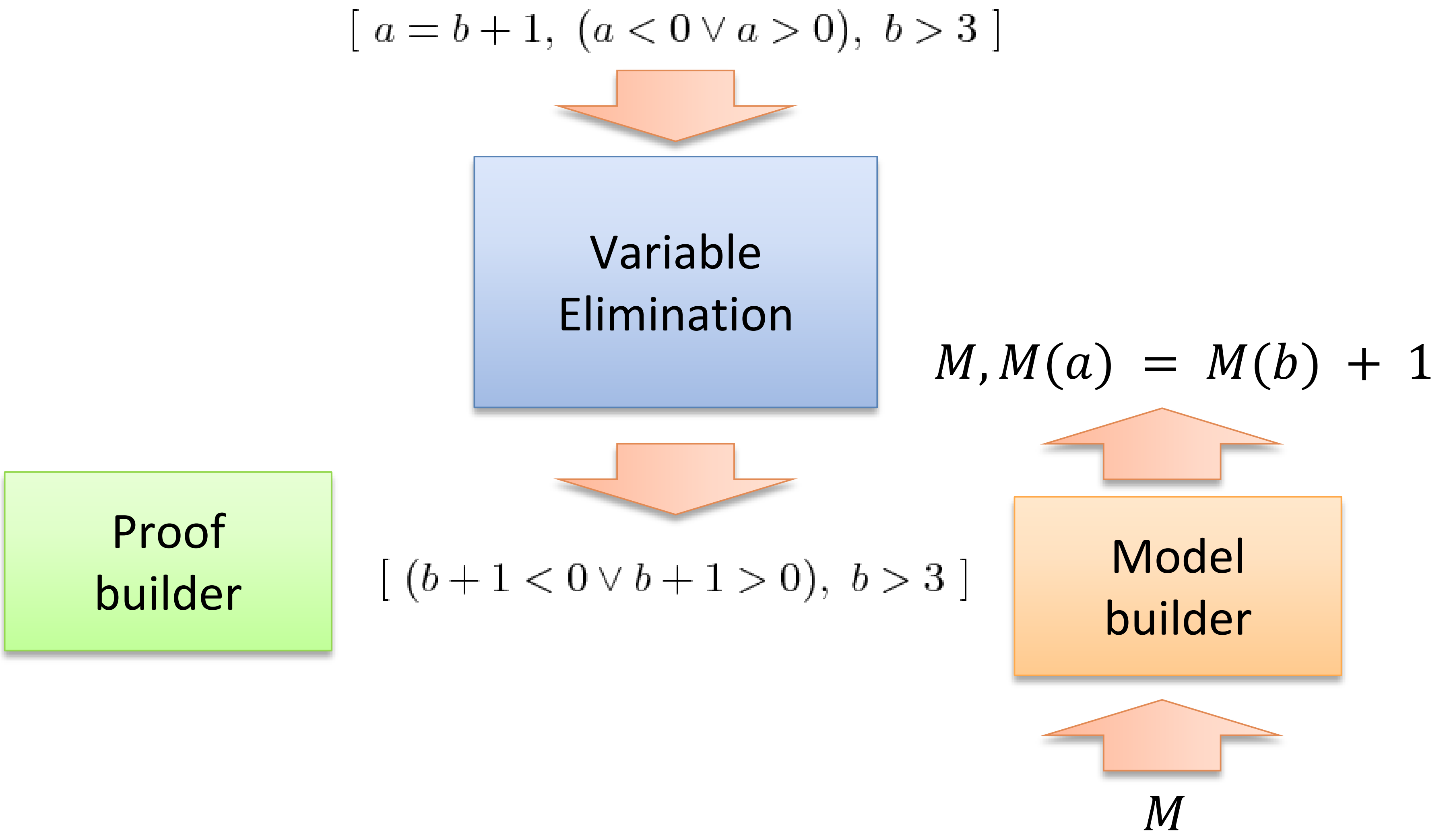
$$x + y + 1 - x - 2 \mapsto y - 1$$

$$p \wedge true \wedge p \mapsto p$$

Example

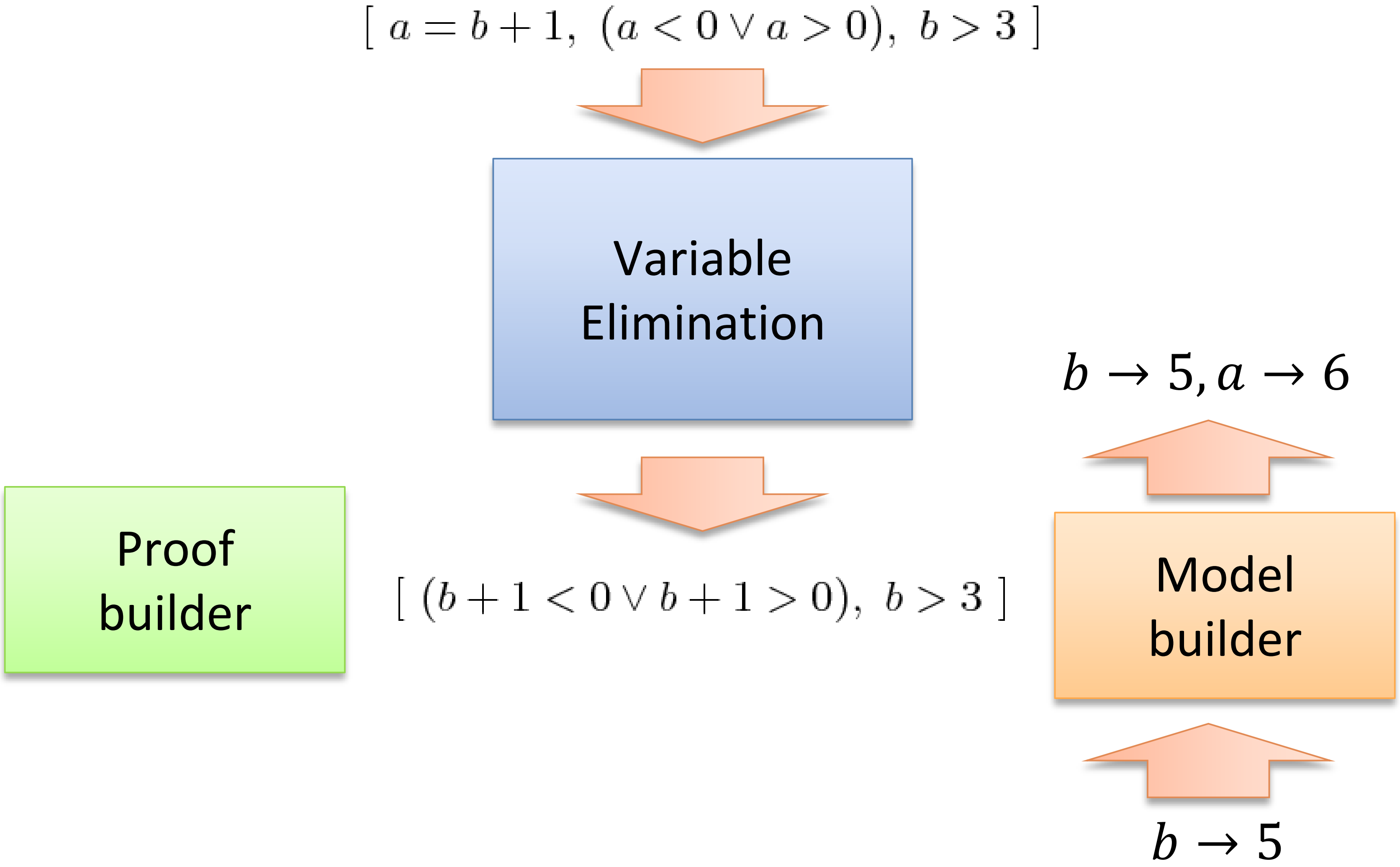

$[\ a = b + 1,\ (a < 0 \vee a > 0),\ b > 3\ ]$
Variable Elimination
Proof builder
$[\ (b + 1 < 0 \vee b + 1 > 0),\ b > 3\ ]$
Model builder

Example



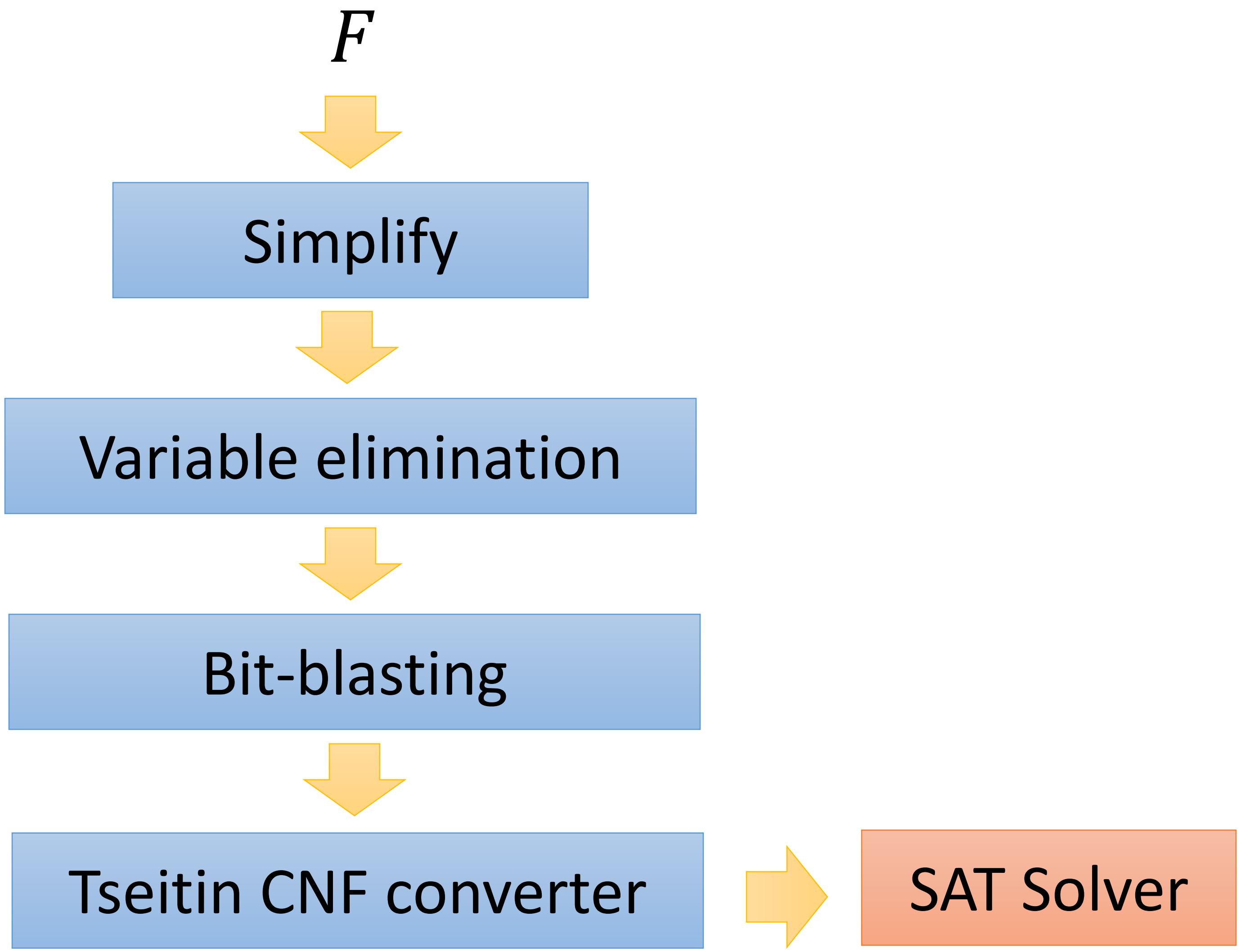
$[\ a = b + 1,\ (a < 0 \lor a > 0),\ b > 3\ ]$
Variable Elimination
$M, M(a) = M(b) + 1$
Proof builder
$[\ (b + 1 < 0 \lor b + 1 > 0),\ b > 3\ ]$
Model builder
M

Example



$[\ a = b + 1,\ (a < 0 \vee a > 0),\ b > 3\ ]$
Variable Elimination
$b \to 5, a \to 6$
Proof builder
$[\ (b + 1 < 0 \vee b + 1 > 0),\ b > 3\ ]$
Model builder
$b \to 5$

Simple QF_BV (bit-vector) solver



Satisfiability Modulo Theories (SMT)

Is formula *F* satisfiable modulo theory *T* ?

SMT solvers have specialized algorithms for *T*

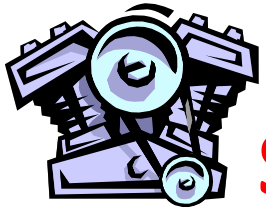
Z3 Solver

DPLL

Simplex

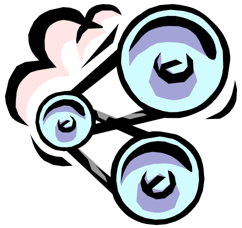
Rewriting

Superposition



Z3 is a collection of

Symbolic Reasoning Engines



Congruence Closure

Groebner Basis

∀∃ elimination

Euclidean Solver

Satisfiability Modulo Theories (SMT)

$$b + 2 = c \text{ and } f(read(write(a,b,3), c-2)) \neq f(c-b+1)$$

Microsoft Research

Satisfiability Modulo Theories (SMT)

$$\boxed{b + 2 = c} \text{ and } f(read(write(a,b,\boxed{3}), \boxed{c-2}) \neq f(\boxed{c-b+1})$$

Arithmetic

Satisfiability Modulo Theories (SMT)

$$b + 2 = c \;\; \text{and} \;\; f(\boxed{read}(\boxed{write}(a,b,3), c-2) \neq f(c-b+1)$$

Array Theory

Microsoft Research

Satisfiability Modulo Theories (SMT)

$$b + 2 = c \text{ and } \boxed{f(}read(write(a,b,3), c-2) \neq \boxed{f(}c-b+1)$$

Uninterpreted Functions

Approaches to linear arithmetic

- Fourier-Motzkin:
  - Quantifier elimination procedure

    $\exists x\, (t \leq ax \wedge t' \leq bx \wedge cx \leq t'') \Leftrightarrow ct \leq at' \wedge ct' \leq bt''$
  - Polynomial for difference logic.
  - Generally: exponential space, doubly exponential time.

- Simplex:
  - Worst-case exponential, but
  - Time-tried practical efficiency.
  - Linear space

Combining Theory Solvers: Nelson-Oppen procedure

Initial state: L is set of literals over $\Sigma_1 \cup \Sigma_2$

Purify: Preserving satisfiability,
convert L into $L' = L_1 \cup L_2$ such that
$L_1 \in T(\Sigma_1, V)$, $L_2 \in T(\Sigma_2, V)$
So $L_1 \cap L_2 = V_{shared} \subseteq V$

Interaction:
Guess a partition of V_{shared}

Express the partition as a conjunction of equalities.
Example, $\{ x_1 \}, \{ x_2 , x_3 \}, \{ x_4 \}$ is represented as:
$\psi: x_1 \neq x_2 \wedge x_1 \neq x_4 \wedge x_2 \neq x_4 \wedge x_2 = x_3$

Component Procedures:
Use solver 1 to check satisfiability of $L_1 \wedge \psi$
Use solver 2 to check satisfiability of $L_2 \wedge \psi$

Example Theory in Z3: Arrays

- Functions: $\Sigma_F = \{ \mathit{read}, \mathit{write} \}$
- Predicates: $\Sigma_P = \{ = \}$
- Convention $a[i]$ means: $\mathit{read}(a,i)$

- Non-extensional arrays T_A:
  - $\forall a, i, v \,.\, \mathit{write}(a,i,v)[i] = v$
  - $\forall a, i, j, v \,.\, i \neq j \Rightarrow \mathit{write}(a,i,v)[j] = a[j]$

- Extensional arrays: $T_{EA} = T_A +$
  - $\forall a, b. ((\forall i.\ a[i] = b[i]) \Rightarrow a = b)$

Suggested reading for JML and contracts

- Paper: ”Design by Contract with JML”.
- JML reference manual (updated occasionally)
- Book: Deductive Software Verification – The KeY Book
  - Chapters 3, 7 and 8 especially. Incrementally introducing more advanced concepts for JML.
- Paper: ”Desugaring JML Method Specifications” for additional help for understanding of JML.
- Relevant papers on chosen tool (SMT Solver or other) by each group.